JMCTest: Automatically Testing Inter-Method Contracts in Java

Paul Börding[1], Jan Haltermann[1], Marie-Christine Jakobs[2(✉)], and Heike Wehrheim[1]

[1] Paderborn University, Paderborn, Germany
[2] LMU Munich, Munich, Germany
M.Jakobs@lmu.de

Abstract. Over the years, Design by Contract (DbC) has evolved as a powerful concept for program documentation, testing, and verification. Contracts formally specify assertions on (mostly) object-oriented programs: pre- and postconditions of methods, class invariants, allowed call orders, etc. Missing in the long list of properties specifiable by contracts are, however, *method correlations*: DbC languages fall short on stating assertions relating methods.

In this paper, we propose the novel concept of *inter-method contract*, allowing precisely for expressing method correlations. We present JMC as a *language* for specifying and JMCTest as a *tool* for dynamically checking inter-method contracts on Java programs. JMCTest fully automatically generates objects on which the contracted methods are called and the validity of the contract is checked. Using JMCTest, we detected that large Java code bases (e.g. JBoss, Java RT) frequently violate standard inter-method contracts. In comparison to other verification tools inspecting (some) inter-method contracts, JMCTest can find bugs that remain undetected by those tools.

1 Introduction

Design by Contract (DbC), first proposed by the Vienna definition language [35], has become a popular concept for documentation, testing, and verification of (mainly) object-oriented software. Today, DbC concepts exist for languages like Eiffel [28], Java (JML [26]), .Net (Code Contracts [19]), C (like used in VCC [17]) or Python [31]. Typically, contracts are directly written into the code and thus also document programs. Contracts are moreover the basis for test generation (e.g., [7,11,14,29,30] for JML) and runtime verification (e.g., [9,13] for JML).

Contracts can refer to different entities of object-oriented programs. Most DbC languages contain pre- and postconditions of methods on normal, i.e. non-exceptional, behavior and class invariants. More sophisticated languages allow to specify history constraints, behavioral subtypes, or type-state properties (call order of methods), incorporate the description of normal as well as exceptional behavior, and include so-called model variables as convenient way of specification. However, all these languages cannot explicitly state method *correlations*.

I. Medina-Bulo et al. (Eds.): ICTSS 2018, LNCS 11146, pp. 39–55, 2018.
https://doi.org/10.1007/978-3-319-99927-2_4

(1) Object o;
true → o.equals(o)

(2) Object o1, o2, o3;
o1.equals(o2) AND o2.equals(o3)
→ o1.equals(o3)

(3) Object o1, o2;
o1.equals(o2) →
o1.hashCode() == o2.hashCode()

(4) MyClass mc;
mc.incr() AND int v = mc.get() →
mc.decr() AND v == mc.get()

```
public class MyClass {
  private int i;
  @Override
  public boolean equals(Object o){
      if (o == null) return false;
      if (o == this) return true;
      if (!(o instanceof MyClass))
            return false;
      return i == ((MyClass) o).i;
  }
  @Override
  public int hashCode(){
    return super.hashCode(); }
  public void incr() {i++;}
  public void decr() {i--;}
  public int get() {return i;}}
```

Fig. 1. Four contracts (left) on the Java method `equals` and the Java methods `incr` and `decr`, respectively, and an example class violating contract 3 (right).

Method correlations describe interactions between methods, i.e., the effects of a method execution on the results of other methods or the relation between method results. Such correlations often exist and constitute an integral part of the (intended) behavior of classes. Even the Java API documentation *informally* states such correlations and expects application classes extending predefined Java classes or implementing Java interfaces to satisfy these. The most prominent one is the contract on the methods `equals` and `hashCode` of class `Object`:

> "If two objects are equal according to the `equals` method, then calling the `hashCode` method on each of the two objects must produce the same integer result." (From: Java API documentation of class `Object`)

While this example concerns a general case (all classes must adhere to this behavior), correlations might also be application specific like one method having the inverse effect of another (e.g. an increment and a decrement). Today's DbC language, however, fall short on specifying method correlations.

In this paper, we rectify this situation by proposing a new type of contract, called *inter-method contract*, that allows to specify method correlations. We propose the language JMC (Java Method Contracts) for writing inter-method contracts for Java. JMC allows to state relations between arbitrary methods of (not necessarily the same) classes. In its syntax, JMC closely follows Java and is thus easy to use for Java programmers. As our running example consider the four contracts given in the left of Fig. 1. The first three contracts specify requirements on the `equals` method: contract (1) states reflexivity, contract (2) transitivity, and contract (3) the above mentioned interplay between `equals` and `hashCode`. Contract (4) specifies that a decrement is the inverse of an increment, a property not expressible in existing DbC languages. All contracts take the form of

an implication (denoted by →): if the first part in the implication is "true", the second part should hold as well. Expressions and method calls in JMC follow standard Java syntax.

With JMCTest we furthermore developed a tool that automatically tests JMC contracts on Java classes. JMCTest is built on Java's reflection mechanism to retrieve constructors of classes under test and to call these as to fully automatically generate objects for test input. Using this input, JMCTest carries out tests evaluating the validity of JMC contracts. Being a dynamic analysis tool, JMCTest aims at finding violations of contracts, not at proving their correctness. For the example class of Fig. 1, JMCTest easily finds a violation of the `equals-hashCode` contract (as the class specializes `equals` to the object variable of the class, but not `hashCode`).

To evaluate the effectiveness of JMCTest, we applied it on real-world production software. Our experiments show that JMCTest can find contract violations in large code bases and detect violations that static analyzers miss.

2 Inter-Method Contracts

Inter-method contracts describe correlations between methods. We next start with presenting the syntax and semantics of inter-method contracts.

Syntax. The BNF-style grammar shown in Fig. 2 sketches the syntax of inter-method contracts like the ones shown in Fig. 1. Terminal symbols are given in quotes, * denotes iteration (including 0 times) and + iteration at least once. To ease contract specification and test generation, we decided to rely on Java syntax for the four non-terminals `VARDECLARATION`, `BOOLEXPR`, `METHODCALL`, and `VARDEFINITION` (thus they are not explicitly specified in the grammar).

An inter-method contract specification consists of a set of inputs followed by the actual contract specification. The inputs of a contract state its *participants* and their types. During testing, different (Java) objects will be generated as concrete participants. The actual contract specification follows the concept of inference rules and consists of a *premise* (the part in front of the arrow) and a *conclusion*. Both, premise and conclusion, consist of a sequence of statement blocks which inspect, manipulate or derive information about objects or classes.[1] We distinguish two types of statement blocks: *predicate blocks* and *function blocks*.

Predicate Blocks. The predicate blocks of a contract determine its validity. More concretely, each predicate block describes a property on the participants of the contract and possible additionally declared entities. To avoid misunderstandings of contracts caused by mistakenly ignoring operator precedence, predicate blocks must not mix conjunction and disjunction.

Function Blocks. In contrast to a predicate block, the task of a function block is limited to changing the state, e.g., to properly initialize or configure objects, and to extract and store information for later usage.

[1] An empty sequence is an abbreviation for the one-element sequence `true`, see contract (1) in Fig. 1.

```
SPECIFICATION ::= INPUTS CONTRACT
INPUTS ::= VARDECLARATION+
CONTRACT := STATEMENTBLOCKS '->' STATEMENTBLOCKS
STATEMENTBLOCKS ::= STATEMENTBLOCK*
STATEMENTBLOCK ::= PREDICATEBLOCK | FUNCTIONBLOCK
PREDICATEBLOCK ::= DISJUNCTION | CONJUNCTION
DISJUNCTION ::= BOOLEXPR | BOOLEXPR 'OR' DISJUNCTION
CONJUNCTION ::= BOOLEXPR | BOOLEXPR 'AND' CONJUNCTION
FUNCTIONBLOCK ::= FUNCTIONAL+
FUNCTIONAL ::= METHODCALL | VARDEFINITION
```

Fig. 2. Grammar for inter-method contracts

As an example, consider contract (3) of Fig. 1. The contract declares two participants (objects `o1` and `o2`). Its premise and conclusion consist of a single predicate block. Also, contract (2) has just *one* predicate block in the premise. The AND operator joins two boolean expressions (BOOLEXPRs), not predicate blocks.

Semantics. Our inter-method contracts use an execution-based semantics, which builds upon the Java semantics. This has the advantage that the semantics is well-known to the user and we avoid a semantic gap between contracts and tests. To execute a contract, we require concrete values for the inputs (participants).

Definition 1. *A* concrete input *for a contract is a function that maps each input variable (participant) of the contract to a value/object of a proper type.*

Given a concrete input for a contract, we can define the semantics of the contract on that concrete input. To this end, we first of all need to define the execution of the contract with the given input. Due to side-effects of e.g. method calls in function blocks, the execution order of statements matters. Our semantics uses a *sequential* execution order that starts with the first block of the premise and ends with the last block of the conclusion. The statements in predicate and function blocks are also executed from left to right. However, there is a difference between the execution of predicate and function blocks. While function blocks only need to be executed, for predicate blocks also the validity of the property checked by that block must be recorded. During testing, the validity may be stored directly in a (boolean) variable or encoded implicitly in the control-flow. Furthermore, a predicate block will be executed lazily, i.e., as soon as its result (boolean value) is fixed, the remaining expressions are not executed. Thus, we use the Java operators `&&` and `||` for conjunction and disjunction during testing.

Next to the execution, we must also define the validity of a contract on that concrete input. Testing aims at finding contract violations. Thus, we define when a contract is *violated*. Since thrown exceptions are ambiguous, they may be thrown because a contract is violated or the contract is improper (e.g. violates method preconditions), we exclude exceptions from our violation definition.

Definition 2. *A concrete input* violates a contract *if (1) all predicate blocks occurring in the premise evaluate to true, (2) at least one predicate block occurring in the conclusion evaluates to false, and (3) the execution terminates normally.*

Note again that ANDs and ORs are not used to join predicate blocks, just boolean expressions. This semantics of contracts is the basis for test generation.

3 Test Generation

The goal of test generation is to automatically build JUnit tests [23] that check whether a given set of classes adheres to a contract. The JUnit tests depend on two main building blocks: (1) checking whether concrete inputs violate contracts and (2) generating the concrete inputs for testing (test input data generation).

Generating Violation Checking Test Code. First of all, we need code that checks whether a concrete input violates the contract. To achieve modularity, we decided to generate a method `testContract` for that check and to provide the concrete input via parameters as e.g. done in parameterized JUnit tests. Input generation itself is done by the second building block. The `testContract` method has the following signature

```
int testContract(list_of_input_types)
```

where `list_of_input_types` is a placeholder for the list of parameters. The list of parameters will contain one parameter for each input variable of a contract.

We use an integer return value instead of a boolean one and no assert statements in the method `testContract` to be able to return some more information about the outcome of the check (0 = premise not fulfilled, 1 = premise and conclusion fulfilled, 2 = contract violated). More specifically, for each contract, we generate a violation check method of the following form. The generation of the parameter list, the premise and the conclusion is contract dependent and explained below.

```
public int testContract(<parameter list>) {
        <premise>
        <conclusion>
        return 1; }
```

Generating the *parameter list* is simple. Since in the `INPUTS` of a contract we use variable declarations without initialization to specify the input variables, we simply turn the `INPUTS` into a parameter list. For the premise code, we need to translate a sequence of statement blocks. The idea is to generate a sequence of Java statements by translating each statement block into a Java statement. Function blocks are easy (since this is already correct Java code): we just take them as they are. In contrast, a predicate block is translated into an if-statement as to capture the semantics of contracts, which crucially depends on the evaluation of predicate blocks. The if-statement takes the following form:

```
if( !(<property>)) {
    return 0; }
```

The if-statement checks whether the property of the predicate block is not fulfilled. In this case, the value 0 is returned, i.e., the contract is not violated on the concrete test input because the premise is already not fulfilled. Note that this is correct since a violation (see Definition 2) requires all predicate blocks in the premise to evaluate to true. The property of a predicate block itself is either a disjunction or a conjunction, and thus translated either using `||` or `&&`.

Generating code for the *conclusion* (second `STATEMENTBLOCKS` element in a contract) is similar to the generation of the premise code. The only difference is the return value, which for the conclusion is 2 when a property is not fulfilled. Figure 3 shows the `testContract` method generated for the `equals-hashCode`.

```
public int testContract(Object o1, Object o2) {
  if(!(o1.equals(o2)) {
    return 0; }
  if(!(o1.hashCode() == o2.hashCode())) {
    return 2; }
  return 1; }
```

Fig. 3. Generated `testContract` method checking violation of `equals-hashCode`

Detecting a contract violation with a single concrete input is unlikely. A contract must be checked with many different concrete inputs. While we could have created one test case per concrete input, we decided to bundle all violation checks for a particular class into *one* JUnit test case and report violation details in a log. This improves the clarity of the test result. The method `testContractImpl` checks such a set of inputs, calling for each input the `testContract` method, and logs the following data.

Logging Test Inputs. For each observed contract violation or exception, the `toString()` representations of all input values is logged. For the first n[2] violations or exceptions, the input values are also serialized to a file associated with the respective violation or exception.

Logging Statistical Data. Besides test inputs, the `testContractImpl` method logs statistical data about the contract checks for the implementation (class). A list of this data can be found in Table 1.[3]

The method `testContractImpl` ends with the JUnit assertion

```
assertTrue(failures == 0 && exceptions == 0);
```

referring to the collected statistical data, i.e., JUnit signals a successful test when no test input violates the contract nor causes an exception to be thrown.

[2] The number n is user-configurable.

[3] The ratio of PremiseFF to Runs is a metric indicating how many of the generated tests are relevant for contract checking.

Table 1. Logged statistics

Name	Description
Runs	Number of executions of method `testContract`
PremiseFF	Number of executions of method `testContract` with fulfilled premise
Failures	Number of executions of method `testContract` with violations
Exceptions	Number of executions of method `testContract` which threw exceptions
FailRate	Failures * 100/PremiseFF

Generating Test Input Data. To execute test cases, we need concrete inputs. We build the concrete inputs from input data given for each input variable. Hence, we need to generate input values for each type occurring in the `INPUTS` of a contract. Note that with respect to coverage, we need not achieve coverage of the `testContract` method, which existing white-box input generators would likely try, but rather of the contract and the methods involved in the contract. Thus, we decided to use an efficient, black box strategy that mainly generates input values randomly. In addition, we allow user guidance to steer or restrict the random generation. Table 2 summarizes the configuration options for test input generation. In our random generation, we distinguish between primitive types, Strings, and other object types.

Table 2. Configuration parameters for test input generation

Type	Options	Description
Int	Sampling	Values for `int`s chosen at random, range and number parameterizable
	Fixed	Values for `int`s user defined
Double	Sampling	Values for `double`s chosen at random, range and number parameterizable
	Fixed	Values for `double`s user defined
String	Sampling	Values for `String`s chosen at random, pattern and number parameterizable
	Fixed	Values for `String`s user defined
Object	Depth	Maximal nesting depth in constructor calls.
	AllowNull	Allow/Disallow `null` as parameter in constructor calls above Depth
	Creation	`Search` (all constructor combinations) or `Random` samples
	#Empty	Number of objects constructed with parameterless (empty) constructor
	#NonEmpty	Number of random objects constructed with other (non-empty) constructors (only if Creation is set to `Random`)

Primitive Data Types. We use a rather standard generation for primitive data types. For boolean types, the test generator uses both values `true` and `false`. In all other cases, the test generator relies on a predefined selection from the range of the data type. The selection depends on the configuration and consists of the fixed set and a number of random values from the sample range.

Strings. Whenever a String value is required, the test generator chooses the value from a predefined set of String literals. This predefined set consists of a set of user-provided Strings and a fixed number of randomly created Strings.

Objects. In contrast to the previous values, arbitrary objects cannot be represented by literals. To create objects, one must call *constructors*. Depending on the constructor, parameter values are also required. For primitive data types and Strings as parameters, the predefined value sets described above are used. All other objects have to be created, which again requires calling constructors and building objects for their parameters. The nesting depth of object creation is set by the user. Beyond that limit, only null values are used to avoid infinite object creation. By default, the test generator does a search, i.e., via Java's *reflection* mechanism it retrieves all available constructors of a class and calls these with all combinations of parameter values available, which is rather exhaustive. To speed up the test process, the user can fix the number of created objects for each input variable with an object type. In this case, the constructors are selected randomly for each object creation. Additionally, the user may add the `null` value and decide if the parameterless constructor should be used multiple times.

JUnit Test Generation. Knowing how to check contract violations and how to generate test input data, we have everything at hand to generate the actual tests. To easily identify the generated tests for a contract, we decided to generate one JUnit test class per contract. The class is named after the contract. The following code skeleton illustrates the structure of the generated JUnit test class.

```
// imports
public class <contract name> {
  // set-up
  // test cases
    // testContract and testContractImpl
    // testImplXXX methods  }
```

The import section ensures that the types, the classes under test, and the JUnit elements are known. The set-up section hard codes the test values for primitive data types and Strings, initializes the object generator, and ensures that for each test case the object generator creates test values for all input objects that are not under test. Furthermore, it sets up the loggers.

The third part adds the test cases. Next to the two methods `testContract` and `testContractImpl` described above, there exists one test case per implementation (class `XXX` under test).[4] Each test case tests whether the respective

[4] Note that we could have used one parameterized JUnit test instead, but we think our solution simplifies the detection of the contract violating implementations.

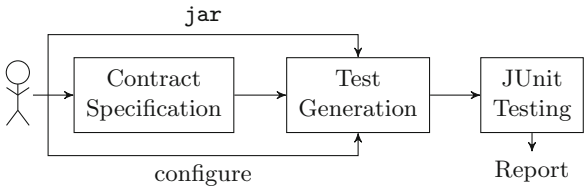

Fig. 4. JMCTest workflow

class under test sticks to the contract. All test cases are defined by the same schema illustrated by the following piece of code.

```
@Test
public void testImplXXX() {
  XXX[] cut =
    (XXX[]) objectGenerator.createInput(XXX.class);
  testContractImpl(...,cut,...); }
```

A test case is described by a `testImplXXX` method. The method is annotated with the `@Test` tag to tell the JUnit framework that the method should be treated as a test case. During its execution, it generates the input values for those input variables that are under test[5]. To this end, it uses the creation method of the object generator. Thereafter, it calls the `testContractImpl` method to test the implementation against the contract. For the parameters, it uses the local array `cut` and the test values defined in the set-up part.

4 Implementation and Evaluation

We briefly explain our implementation of this testing framework (called JMCTest) and report on the results of our experimental evaluations.

4.1 JMCTest

JMCTest is a research prototype written in Java that supports specification and automatic testing of inter-method contracts for Java.

Figure 4 describes the workflow of JMCTest. The user starts with the contract definition using the graphical contract editor of JMCTest. Thereafter, she configures and starts the test generation for that contract. In her configuration, she specifies how to generate input data. Additionally, she provides a `jar`-file that defines which class implementations to test. Based on the given configuration, JMCTest automatically creates a set of JUnit tests, one for each class in the `jar`-file. After all tests are generated, they are executed with the JUnit 4 Framework [22]. Finally, the number of executed and failed test cases as well

[5] Currently, all input variables under test must have the same object type.

as the failed test cases together with their failure reason are reported. Detailed information about the executed and failed test cases are provided in the log files.

4.2 Evaluation

We carried out a number of experiments to evaluate the effectiveness and efficiency of JMCTest. Since our JMC language contains a large part of Java's expression language, we easily expressed the informal Java API contracts as well as all contracts from our projects in JMC. Writing contracts in JMC is effective.

Comparison with Other Tools. For the evaluation, we wanted to compare JMCTest with other analysis tools (for Java). As there have been no inter-method contracts before, the number of tools checking a comparable sort of properties is limited. Model checkers like Java PathFinder [34] are designed to check specific properties of programs and thus cannot directly be used for inter-method contracts. Nevertheless, we found two categories of tools that principally can check at least some form of inter-method contracts: (a) property-based testing tools (similar to QuickCheck [15]) and (b) bug pattern detection tools. To decide against which tools to compare JMCTest, we took a closer look at some tools and evaluated them with respect to the following criteria:

- for testers only: the ability to automatically generate test input, for primitive data types as well as for objects,
- the ability to check arbitrary contracts, possibly via an encoding of them in a different form (e.g., property),
- the ability to work on a `jar`-file and test *all* classes in it,
- the ability to return all errors found (vs. stop with the first error found).

In the first category, we chose the property-based tester JCheck[6] found at GitHub. There were more options available, but none seemed to be frequently used. For the second category, we chose three tools: EqualsVerifier[7], which is specifically tailored to properties of the `equals`-method, FindBugs [2,22], a static analysis tool frequently employed today (e.g. also at Google) to find common bug patterns in Java programs, and RANDOOP [30], a feedback-directed random test generator. An alternative for FindBugs could have been PMD[8], but we decided not to have two tools using the same basic checking principle. Table 3 provides a summary of the evaluation. Due to their limitations e.g. on input generation, JCheck and EqualsVerifier are improper for automatic contract checking. Thus, we only compare JMCTest with FindBugs and RANDOOP.

[6] http://www.jcheck.org/.
[7] https://github.com/jqno/equalsverifier.
[8] https://pmd.github.io/.

Table 3. Functionality of tools considered for the comparison

Tool	Input generation		Arb. contracts	`jar`	All errors
	Primitive data type	Object			
JCheck	✓	×	✓	×	✓
EqualsVerifier	✓	×	×	×	×
FindBugs	n.a.	n.a.	×	✓	✓
RANDOOP	✓	✓	×	✓	✓
JMCTest	✓	✓	✓	✓	✓

Claims. Besides showing that it is feasible to use JMCTest for testing inter-method contracts, our experiments aimed at evaluating the following claims.

Claim 1 JMCTest can find real violations of inter-method contracts.
Claim 2 JMCTest can find violations that other tools do not detect.
Claim 3 JMCTest is fast enough to be integrated into the build process.

For the evaluation of these claims, we planned the following experiments.

Claim 1. We used JMCTest to check the `equals-hashCode` contract on three real-world software projects: (1) CPAchecker (a software analysis tool, [5]), (2) JBoss (a J2EE middleware framework) and (3) Java rt.jar (the Java system library). We restricted ourselves to the `equals-hashCode` contract since it is universally applicable to all Java source code and Java programmers are trained to follow this contract. Thus, violations of the contract constitute a bug.

Claim 2. To evaluate claim 2, we compare JMCTest against FindBugs and RANDOOP. FindBugs checks for specific patterns, e.g., classes overriding either `equals` or `hashCode`, to detect violations of the `equals-hashCode` contract. RANDOOP generates random sequences of constructor and method calls and e.g. checks that the resulting objects meet the `equals-hashCode` contract.

Claim 3. For the evaluation of the last claim, we run JMCTest on the three software projects with different configurations (in particular, differences in the number of objects constructed). In the experiments, we wanted to see the runtime and the number of objects to be constructed until bug finding converges, i.e., until no more bugs are found when the number of test cases is increased. This helps to see whether it is possible to run JMCTest during a nightly build.

Results. We performed our experiments on a machine with an Intel i5-300HQ v6 CPU with a frequency of 2.3 GHz, 16 GB of memory, and a Windows 10 operating system. Execution times are reported in seconds. Note that a ground truth for the experiments, i.e., the real number of violations of the `equals-hashCode` contract in the three software projects, is not known.

Table 4. Overview of the JMCTest analysis

Software	Tested classes	Reported errors	#NonEmpty objects	Depth
CPAchecker	316	3	50	3
JBoss	214	9	50	3
Java rt.jar	1088	124	100	3

Claim 1. For all three software projects, Table 4 shows the number of classes[9] tested against the `equals-hashCode` contract, the number of contract violations found as well as the number of objects constructed during the test using *non-empty* constructors (i.e., not the parameterless constructor) and the nesting depth of object creation. We see that JMCTest detects contract violations in all three projects, even in the Java runtime library (Java rt.jar). Violations in Java rt.jar were found in the package `com.sun.org.apache.bcel.internal.generic`.

Claim 2. We let FindBugs, RANDOOP, and JMCTest analyze the same classes and compared the number of reported violations. Figure 5 shows the results of this comparison in two bar charts. Each bar chart shows for every analyzed software project[10] the number of violations reported by FindBugs (RANDOOP), by JMCTest, by both (i.e., the intersection of bugs found), and by JMCTest only (i.e., bugs found by JMCTest, but not by the other tool). We see that JMCTest always finds some violations that are not detected by the other tool (rightmost bars). FindBugs and RANDOOP always find more violations than JMCTest. A manual inspection of some randomly chosen warnings revealed that many of FindBugs' violations are false warnings. Some real bugs are, however, missed by JMCTest. Real bugs reported by FindBugs are missed because JMCTest fails to construct objects of the classes under test due to (1) abstract classes (no object construction possible) and (2) missing access permissions, which disallowed object construction. As such problems are inherent to our technique (testing needs object creation and method execution), we see no way of circumventing them. For two bugs reported by RANDOOP, JMCTest's random input generator did not build suitable inputs, although in principle it could. The other three are missed because JMCTest cannot deal with generics and does not test the `equals-hashCode` contract with objects of different classes.

Claim 3. Finally, for claim 3 we were interested in runtimes of JMCTest. To be practically usable, testing results should be obtained in a reasonable amount of time. Figure 6 shows the results for the three software projects. On the x-axis the number of non-empty constructor calls made during the tests is given. The y-axis gives the runtimes in seconds. For both CPAchecker and Java rt.jar,

[9] We tested all public classes which did not use `Object.equals`.

[10] We compare RANDOOP and JMCTest only on the CPAchecker project because RANDOOP failed on `Java rt.jar` and got stuck on `JBoss`.

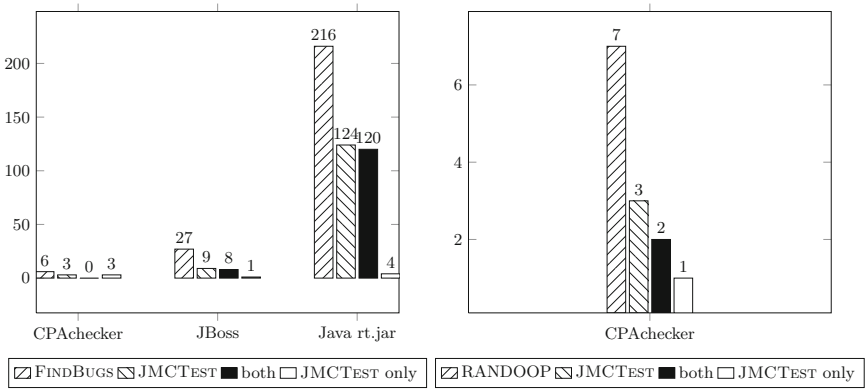

Fig. 5. Comparing JMCTest with FindBugs (left) and RANDOOP (right)

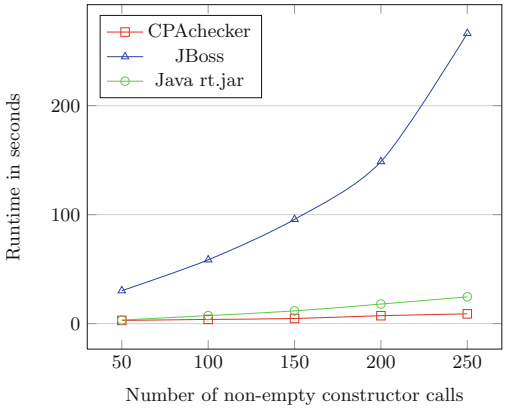

Fig. 6. Runtimes of JMCTest

test results are obtained within some seconds. For JBoss, runtimes are higher. This is due to the complicated nature of constructors in JBoss, which for instance have to bind ports. We were also interested in finding out how many constructor calls are necessary to find all bugs: for CPAchecker and JBoss the number of bugs remains stable after testing with 50 constructor calls, for Java rt.jar it is 100 constructor calls. More experiments are, however, needed to find out what a good value for the #NonEmpty configuration parameter is. Nevertheless, even if we set this configuration parameter to 150 or 200, JMCTest finished after a few minutes. Thus, JMCTest is fast enough to run as part of the build process.

In summary, the experiments show that JMCTest can be practically applied, also on large code bases, and can detect contract violations, which other tools do not detect.

5 Related Work

Behavioral Contract Languages. One of the first contract language is the Eiffel language [28]. Nowadays, various contract languages exist. All of them are either embedded contracts or contracts defined by an additional language. Embedded contract languages like e.g. jContractor [24], TreatJS [25], the Spec# programming language [3] or the conditional properties used by QuickCheck [15] directly write contracts in the programming language itself. In contrast, contracts like Jass [4], the Java modeling language (JML) [26], Praspel [18], and the VCC annotated C [17] are defined in an additional language, whose sole purpose is to specify the respective contract. Nevertheless, the mentioned contracts are often embedded in the comments of the source code. We also developed a separate contract language. Instead of pre-, postconditions, invariants, protocols or refinement, our language tackles inter-method relations. The only other contract languages that can express inter-method relations are conditional properties [15] and parametrized unit tests [33], which express properties by arbitrary function code. In contrast to our approach, both mix specification and testing code, which impairs the developer's access to inter-method contracts.

A different type of contracts are algebraic specifications of abstract data types [21]. Algebraic specifications use algebraic equations, the contracts, to state relationships among operations of abstract data types. Our inter-method contracts cover these relations, but allows us to specify relations beyond abstract data types, e.g., relations between methods of different classes.

Checking Behavioral Contracts. Techniques like static analysis [1,3,8], runtime verification [3,4,18,20,26], or testing [7,11,14,16,27,29,30,32,36] are used to check behavioral contracts. Next, we focus on testing, the technique we apply.

Few test approaches [16,32] automatically test fixed contracts. JCrasher [16] creates tests to inspect the robustness of public methods. Pradel and Gross [32] test substitutability of subclasses. Both randomly sample primitive values. Constructors and methods with a proper return type are used to create objects. JMCTest checks user-defined contracts, also uses random primitive values in addition to fixed values, but only uses constructors for object creation.

Like us, many test approaches, e.g., [7,11,12,27,29,30,36], use the contract specification to generate a test oracle, which decides if a test fails or passes. Those approaches, however, build the oracle from class invariants and a method's pre- and postconditions and differ in their input generation.

The JML-JUnit framework [11] is semi-automatic and generates tests based on user-specified inputs. Bouquet et al. [6] build tests that cover all parts of a (method's) specification. Often, test inputs are generated randomly. Jartege [29] randomly creates sequences of method calls to generate input objects. Similarly, JET [12] applies a random approach in which input objects are created by a constructor call followed by a sequence of method calls mutating the object. Both, Jartege and JET rely on JML specifications. In contrast, RANDOOP [30] and ARTGen [27] check contracts provided by classes implementing specific interfaces. RANDOOP [30] builds a test case concatenating randomly chosen existing sequences and extending them with a random method call. ARTGen [27]

performs adaptive random testing [10], i.e., it selects the test case from a pool of test inputs that is farthest away from the already used test cases. The pool is modified randomly, calling methods on existing objects or adding new objects created from random method sequences. QuickCheck [15] tests properties defined by functions in the code. Test cases are created randomly with (user-defined) generators. Parametrized unit tests [33], unit test methods with parameters similar to our `testContract` method, check properties defined in the test method code. Often, test inputs are generated with symbolic execution to execute the parametrized unit tests and to achieve high code coverage.

In contrast to JMCTest, Korat [7] and JMLAutoTest [36] systematically explore a bounded search space. Korat [7] uses a finitization code that describes the search space and a predicate checking if an input is valid. JMLAutoTest [36] also relies on finitization code. Provided with a finitization and a JML specification, it systematically generates all non-isomorphic test cases, excluding those which violate class invariants, and checks them against the JML specification

JMCTest uses the inter-method contract as test oracle and generates inputs from random samples and fixed inputs. However, it is limited to constructor calls for object generation. While conditional properties [15] and parametrized unit tests [33] subsume our inter-method contracts, in contrast to QuickCheck and parametrized unit tests, JMCTest is fully automatic.

6 Conclusion

For more than 20 years, languages and tools have been developed to support the idea of Design by Contract. Regardless, existing languages and tools mostly focus on pre-, postconditions, and invariants. This makes it difficult if not impossible to state and check contracts that focus on correlations between methods.

Our inter-method contract language offers a mechanism to formally describe method correlations and, thus, enables their automatic validation. Due to its similarity to Java, our language is easy to learn. Furthermore, we did not stop at the language level, but we carried on with tool support for specification and validation of inter-method contracts. Our prototype tool JMCTest provides a user interface for inter-method contract specification. It can automatically test a set of implemented classes against a specified inter-method contract. Although JMCTest is an academic prototype, it already detected real violations of the `equals-hashCode` contract in existing, well-maintained software projects. More impressively, some of these violations have not been found by other tools.

References

1. Ahrendt, W., et al.: The KeY platform for verification and analysis of Java programs. In: Giannakopoulou, D., Kroening, D. (eds.) VSTTE 2014. LNCS, vol. 8471, pp. 55–71. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12154-3_4
2. Ayewah, N., Hovemeyer, D., Morgenthaler, J.D., Penix, J., Pugh, W.: Using static analysis to find bugs. IEEE **25**(5), 22–29 (2008). https://doi.org/10.1109/MS.2008.130

3. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30569-9_3
4. Bartetzko, D., Fischer, C., Möller, M., Wehrheim, H.: Jass - Java with assertions. Electr. Notes Theor. Comput. Sci. **55**(2), 103–117 (2001). https://doi.org/10.1016/S1571-0661(04)00247-6
5. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
6. Bouquet, F., Dadeau, F., Legeard, B.: Automated boundary test generation from JML specifications. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 428–443. Springer, Heidelberg (2006). https://doi.org/10.1007/11813040_29
7. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on Java predicates. In: ISSTA, pp. 123–133. ACM (2002). http://doi.acm.org/10.1145/566172.566191
8. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: advanced specification and verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_16
9. Chalin, P., Rioux, F.: JML runtime assertion checking: improved error reporting and efficiency using strong validity. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 246–261. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68237-0_18
10. Chen, T.Y., Leung, H., Mak, I.K.: Adaptive random testing. In: Maher, M.J. (ed.) ASIAN 2004. LNCS, vol. 3321, pp. 320–329. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30502-6_23
11. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: the JML and JUnit way. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 231–255. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-47993-7_10
12. Cheon, Y.: Automated random testing to detect specification-code inconsistencies. In: Karras, D. (ed.) Conference on Software Engineering Theory and Practice, pp. 112–119. International Society for Research in Science and Technology (2007)
13. Cheon, Y., Leavens, G.T.: A runtime assertion checker for the Java Modeling Language (JML). In: Conference on Software Engineering Research and Practice, pp. 322–328. CSREA Press (2002)
14. Cheon, Y., Rubio-Medrano, C.E.: Random test data generation for Java classes annotated with JML specifications. In: Arabnia, H.R., Reza, H. (eds.) Conference on Software Engineering Research and Practice, pp. 385–391. CSREA Press (2007)
15. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: ICFP, pp. 268–279. ACM (2000). http://doi.acm.org/10.1145/351240.351266
16. Csallner, C., Smaragdakis, Y.: JCrasher: an automatic robustness tester for Java. Softw. Pract. Exper. **34**(11), 1025–1050 (2004). https://doi.org/10.1002/spe.602
17. Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte, W.: VCC: contract-based modular verification of concurrent C. In: ICSE, pp. 429–430. IEEE (2009). https://doi.org/10.1109/ICSE-COMPANION.2009.5071046

18. Enderlin, I., Dadeau, F., Giorgetti, A., Ben Othman, A.: Praspel: a specification language for contract-based testing in PHP. In: Wolff, B., Zaïdi, F. (eds.) ICTSS 2011. LNCS, vol. 7019, pp. 64–79. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24580-0_6
19. Fähndrich, M.: Static verification for code contracts. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 2–5. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15769-1_2
20. Fähndrich, M., Barnett, M., Logozzo, F.: Embedded contract languages. In: SAC, pp. 2103–2110. ACM (2010). http://doi.acm.org/10.1145/1774088.1774531
21. Guttag, J.V., Horning, J.J.: The algebraic specification of abstract data types. Acta Informatica **10**(1), 27–52 (1978). https://doi.org/10.1007/BF00260922
22. Hovemeyer, D., Pugh, W.: Finding bugs is easy. SIGPLAN Not. **39**(12), 92–106 (2004). http://doi.acm.org/10.1145/1052883.1052895
23. JUnit: Junit4 (2012–2017). http://junit.org/junit4/
24. Karaorman, M., Hölzle, U., Bruno, J.: jContractor: a reflective java library to support design by contract. In: Cointe, P. (ed.) Reflection 1999. LNCS, vol. 1616, pp. 175–196. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48443-4_18
25. Keil, M., Thiemann, P.: TreatJS: higher-order contracts for JavaScripts. In: Boyland, J.T. (ed.) ECOOP. LIPIcs, vol. 37, pp. 28–51. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2015). https://doi.org/10.4230/LIPIcs.ECOOP.2015.28
26. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Softw. Eng. Notes **31**(3), 1–38 (2006). http://doi.acm.org/10.1145/1127878.1127884
27. Lin, Y., Tang, X., Chen, Y., Zhao, J.: A divergence-oriented approach to adaptive random testing of Java programs. In: ASE, pp. 221–232. IEEE (2009). https://doi.org/10.1109/ASE.2009.13
28. Meyer, B.: Design by contract: the Eiffel method. In: TOOLS, p. 446. IEEE (1998). https://doi.org/10.1109/TOOLS.1998.711043
29. Oriat, C.: Jartege: a tool for random generation of unit tests for Java classes. In: Reussner, R., Mayer, J., Stafford, J.A., Overhage, S., Becker, S., Schroeder, P.J. (eds.) QoSA/SOQUA -2005. LNCS, vol. 3712, pp. 242–256. Springer, Heidelberg (2005). https://doi.org/10.1007/11558569_18
30. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: ICSE, pp. 75–84. IEEE Computer Society (2007). http://doi.acm.org/10.1145/1297846.1297902
31. Ploesch, R.: Design by contract for Python. In: APSEC, pp. 213–219. IEEE (1997). https://doi.org/10.1109/APSEC.1997.640178
32. Pradel, M., Gross, T.R.: Automatic testing of sequential and concurrent substitutability. In: ICSE, pp. 282–291. IEEE (2013). https://doi.org/10.1109/ICSE.2013.6606574
33. Tillmann, N., Schulte, W.: Parameterized unit tests. In: FSE, pp. 253–262. ACM (2005). http://doi.acm.org/10.1145/1081706.1081749
34. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. **10**(2), 203–232 (2003). https://doi.org/10.1023/A:1022920129859
35. Wegner, P.: The Vienna definition language. ACM Comput. Surv. **4**(1), 5–63 (1972). http://doi.acm.org/10.1145/356596.356598
36. Xu, G., Yang, Z.: JMLAutoTest: a novel automated testing framework based on JML and JUnit. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, pp. 70–85. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24617-6_6